第3章

操作系统的启动

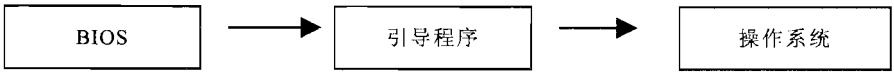
操作系统的启动与硬件体系结构密不可分。一个操作系统究竟应该按照什么样的方式启动并没有一套统一的标准。因此我们可以说，操作系统的启动过程是一个与操作系统自身无关的过程，但却是一个与硬件体系结构相关的过程。

这句话听起来不好理解，却能够恰当表达操作系统启动的本质。举个例子来说，同样是 Linux 操作系统，在 x86 体系结构下与在 ARM 体系结构下启动是完全不同的，这就是所谓的“启动过程与操作系统本身无关”，而 Windows 和 Linux 虽然分属不同的操作系统，但如果两个操作系统都在 x86 下运行，其启动过程从结构上来讲仍然是类似的，这就是所谓的“启动过程与硬件体系结构相关”。

但是，这里似乎忽略了一个很重要的概念——操作系统应该怎样界定，或者说操作系统是从上电那一刻开始算起呢还是从其他时刻开始算起？想要说清楚这个问题，我们首先来讨论一下操作系统的启动流程。

3.1 启动流程

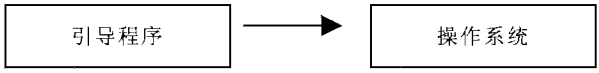
让我们先从计算机说起，计算机的启动流程分为如下三个阶段：



BIOS（Basic Input/Output System）程序是指固化到主板上 BIOS 芯片内的一段初始启动程序，是系统启动时运行的真正意义上的第一段程序。在计算机中，BIOS 程序主要负责硬件自检和初始化、中断处理程序的基本设置，以及从存储器中加载零道零扇区的代码到内存并从该位置开始运行。而存储在外存零道零扇区的代码，通常就是所谓的引导程序了。

简而言之，引导程序就是用来引导操作系统的。由于多数操作系统都要存储在外存中而运行于内存。因此，需要一段小程序能够将操作系统从外存搬运到内存之中并交出 CPU 的控制权，这就是引导程序存在的意义。引导程序与操作系统有很大的关系，有些特定的引导程序只能引导特定的系统，但他们仍然不是操作系统本身。引导程序的作用就是给被引导的操作系统提供运行时所需的环境和条件，然后从外存中把真正的操作系统加载到内存中运行。

以上就是一个普通计算机的启动过程，但这样的过程并不一定适合于嵌入式系统。在很多嵌入式平台中，人们会将计算机的三个启动阶段简化为两个：



这里，引导程序通常会固化到 Flash 或 ROM 存储器中，当系统上电时会首先被运行，起到初始化基本硬件和引导操作系统的双重作用。

还有一些更加简单的嵌入式操作系统，不需要额外的引导程序就可以直接运行对硬件的初始化等工作，作为该操作系统的一部分而发挥作用。

以上是有关操作系统启动的最简要的介绍，有些读者可能会产生如下疑问，如果 BIOS 或引导程序已经对板载硬件完成了初始化，操作系统在被引导以后是否需要重复执行这个工作？例如，引导程序已经完成了串口设备的初始化工作，在操作系统运行时，是否需要重新初始化串口？

这个问题的答案不言而喻，操作系统是有必要对所有硬件重新进行初始化的。究其原因，可以总结为一点：我们不能让操作系统只适应于特定的引导程序，或者可以说我们希望任何引导程序都能引导我们的操作系统。但引导程序五花八门，有的简单，有的复杂，初始化过程又各不相同，如果将启动时需要进行的初始化任务全权交给引导程序负责，势必会降低操作系统的

通用性，容易产生错误。因此比较保险的做法是，无论引导程序是如何初始化的，操作系统自身的初始化过程都要重新进行。

既然如此，那么就可以按照上述原则对操作系统从何处开始这一问题做界定了。我们的操作系统开始于一个基础代码可以运行的环境，处在板载硬件未被初始化的状态，也就是说，我们默认已有一个引导程序保证了我们的操作系统能够出现在内存中并且正确运行，但该引导程序不负责环境的任何初始化工作。

3.1.1 ARM 的启动过程

接下来我们就来聊聊 ARM 的启动。

ARM 的启动其实再简单不过了。在 CPU 上电的时候，ARM 就会首先从地址零处开始运行，第 2 章的几个例子程序都说明了这一点。因为是精简指令集，同时硬件结构较新，所以不需要考虑让人挠墙的历史兼容问题。这些都使程序在ARM体系结构下的启动变得简洁而自然。但是作为操作系统，我们不能简简单单地让程序能够运行就了事了。操作系统的根本目的之一是维护和保障其他程序能够依托本系统顺利工作。因此，区别于简单的裸机程序，操作系统在启动时应至少关心如下两个方面。

1．程序运行栈的初始化。程序的运行离不开栈，程序运行时所需的临时变量、数据、函数间调用的参数传递等都需要栈的支持。如果是在操作系统之上进行编程，堆栈的问题往往不需要程序员关心，但对于操作系统本身而言，需要对用户应用程序提供统一的运行环境和资源，堆栈的处理就不可避免了。因此操作系统在启动时，必须首先解决好堆栈的初始化问题。

2．处理器及外设的初始化。一个操作系统能够正常运行，处理器的正确配置和外设的正常工作也是必不可少的先决条件。例如，中断处理程序能够提供给操作系统非顺序的、突发的执行逻辑。时钟设备则可以产生固定频率的时钟驱动整个系统运行。所以在操作系统各功能模块得以发挥作用之前，处理器及外设也必须被初始化。

上述两条原则只是理论上的，不同的操作系统运行在不同的平台上，实现方法可能各不相同，但无论如何这两条原则都是要遵守的。

3.1.2　ARM 操作系统解读

接下来我们来看一段例子，研究一下 ARM 体系结构下的操作系统在初始化的过程当中，应该如何实现。

代码 3-1

```
.section .startup,"ax"
        .code 32
        .align 0

    b     _start              /* reset - _start          */
    ldr   pc, _undf           /* undefined - _undf       */
    ldr   pc, _swi            /* SWI - _swi              */
    ldr   pc, _pabt           /* program abort - _pabt   */
    ldr   pc, _dabt           /* data abort - _dabt      */
    nop                       /* reserved                */
    ldr   pc, [pc,#-0xFF0]    /* IRQ - read the VIC      */
    ldr   pc, _fiq            /* FIQ - _fiq              */

_undf:  .word __undf          /* undefined               */
_swi:   .word vPortYieldProcessor   /* SWI               */
_pabt:  .word __pabt          /* program abort           */
_dabt:  .word __dabt          /* data abort              */
_fiq:   .word __fiq           /* FIQ                     */

__undf: b  __undf             /* undefined               */
__pabt: b  __padt             /* program abort           */
__dabt: b  __dabt             /* data abort              */
__fiq:  b  __fiq              /* FIQ                     */
```

这一段代码节选自开源嵌入式操作系统 freeRTOS 的初始化部分，与该段代码类似的初始化方式广泛应用于支持 ARM 的各种类型的操作系统和引导程序中，因此非常具有代表性。

代码一开始使用了三条伪指令，其中，.align 的含义和用法我们已经介绍过了（参见第 2 章）。

.section 的作用是声明接下来的代码属于该段，其格式为.section name[, " flags "]或.section name[, subsection]。代码 3-1 中采用的是第一种形式，将整段代码编译到 startup 段中，该段也是整个操作系统的入口。

而方括号内的内容是可选的，代码中该标志被定义为 ax，其中的“a”代表可重定位（section is allocatable），而“x”的意思是可执行（section is executable）。

另一个伪指令.code 32 的作用是编译生成 32 位指令集的代码。.code 的另一种形式是.code 16，其作用是生成 16 位指令集的代码。伪指令.arm 以及.thumb 分别与.code 32 和.code 16 效果相同。我们知道，在传统的 ARM 体系结构中既可以使用 32 位的指令，又可以使用 16 位的指令，此处正好与 ARM 的这套指令系统相统一。

接下来，代码 3-1 将执行该程序的第一条指令——跳转指令，通过该指令程序将直接跳转至_start 处继续运行。既然如此，紧接着跳转指令的其他指令在什么情况下才能运行呢？熟悉 ARM 体系结构的读者可能清楚，包括第一条指令在内，这段代码正是异常向量表。也就是说，所有其他指令都会在异常模式发生时才会运行。

对 ARM 不了解的读者可能会问，异常模式是何物？简单地说，异常模式就是程序由于某种原因执行不正常时，系统所处于的一种运行模式。当异常模式出现时，预先指定的代码将会被运行，从而可以尽可能地解决异常并返回到正常模式中。

在 ARM v6 及以前的体系中定义了七种模式，分别是管理模式（SVC）、快速中断模式（FIQ）、中断模式（IRQ）、未定义模式（UND）、终止模式（ABT）、用户模式（USR）以及系统模式（SYS）。其中，前五种就是所谓的异常模式。每一种异常模式都拥有私有的堆栈指针寄存器 R13、返回地址寄存器 R14 以及模式备份寄存器 SPSR。一段代码只有处在该异常模式下时才有权力访问属于该异常模式的这三个私有寄存器。当某种异常发生时，CPU 会立刻停止当前正在执行的动作，转而去运行预先指定的代码，同时，系统会切换到特定模式中，这是一种被动的模式切换方法。除此之外，程序员还可以通过编程主动进行模式切换，使用的方法是更改 CPSR 寄存器中的模式位。有关异常模式的细节我们将会在第 5 章详细阐述。

既然程序在启动时，就立刻跳转到_start 处运行，那么在_start 位置上 freeRTOS 又做了哪些工作呢？我们接着来看_start 处的代码。

代码 3-2

```
_start:
    ldr   r0, .LC6
    msr   CPSR, #MODE_UND|I_BIT|F_BIT

    mov   sp, r0
    ……
    sub   r0, r0, #SVC_STACK_SIZE
    msr   CPSR, #MODE_SYS|I_BIT|F_BIT /* System Mode */
    mov   sp, r0

    msr   CPSR, #MODE_SVC|I_BIT|F_BIT

    /* Clear BSS. */

    mov     a2, #0          /* Fill value */
    mov     r12, a2         /* Null frame pointer */
    mov     r7, a2          /* Null frame pointer for Thumb */

    ldr     r1, .LC1        /* Start of memory block */
    ldr     r3, .LC2        /* End of memory block */
    subs    r3, r3, r1      /* Length of block */
    beq     .end_clear_loop
    mov     r2, #0

.clear_loop:
    strb    r2, [r1], #1
    subs    r3, r3, #1
    bgt     .clear_loop

.end_clear_loop:
……
    mov     r0, #0          /* no arguments  */
    mov     r1, #0          /* no argv either */

    bl      main
    .LC1:
    .word   __bss_beg__
    .LC2:
.word   __bss_end__
……
    .LC6:
    .word   __stack_end__
```

代码 3-2 从_start 处开始，一步步进行栈指针和 bss 段的初始化工作，最终通过一条 bl 指令跳转到主程序处继续运行。示例代码删减了相对重复和不具有代表性的内容，以使代码结构更清晰。

程序一开始，将局部变量.LC6 的值加载到寄存器中，也就是__stack_end__这一变量的值。从名字上我们可以判断出，该值应该是程序堆栈的顶端。问题似乎随之而来，从理论上讲，任何 CPU 都必须使用一个专有寄存器来指向程序当前的堆栈地址。在 ARM 中寄存器 R13 充当了这一角色，在 x86 下，我们可以通过 ESP 寄存器实现类似的功能。无论什么体系结构，在处理堆栈指针寄存器时都必须面对两个问题：堆栈如何增长？堆栈为满堆栈还是空堆栈？

图 3-1 直观地描述了这两个问题的四种组合情况，所谓堆栈如何增长，指的是堆栈指针更新时，是向着高地址方向还是低地址方向，而所谓满堆栈和空堆栈，指的是当程序利用堆栈指针向堆栈中存储数据或从堆栈中提取数据时，是先将数据存入或读出再更新堆栈指针还是先更新堆栈指针，然后再存取数据，也就是说，当前堆栈指针是否指向了有效数据。因此，堆栈的处理方法可以总结为四种，分别为空递增堆栈、满递增堆栈、空递减堆栈、满递减堆栈。读者可以将这四种处理方法在图 3-1 中一一对号入座。

其实无论堆栈的处理有多少种方法，只要保证所有的函数在保存局部变量或相互调用过程中始终使用同一种处理方法，程序的运行就不会出问题。因为堆栈的使用在汇编语言下是可控的，也就是说，数据向堆栈的存取和堆栈指针寄存器的更新是需要程序员自行处理的。因此，如果所有的程序都使用汇编写成，我们可以直接选择一种处理方式来处理堆栈，就不会出问题了。但如果使用其他语言编程，堆栈的处理已经被编译器屏蔽，程序员并不知道编译器选择了哪种方式，这样编译出来的程序就会存在隐患。例如，使用 C 语言编程，并且编译器是按照满递减的方式处理源代码的，但我们却错误地在操作系统初始化时将堆栈指针寄存器设置成内存的最低地址，一旦该程序使用堆栈，便会出现数据异常的情况。

那这个问题该如何解决呢？答案是，通过过程调用标准来解决。

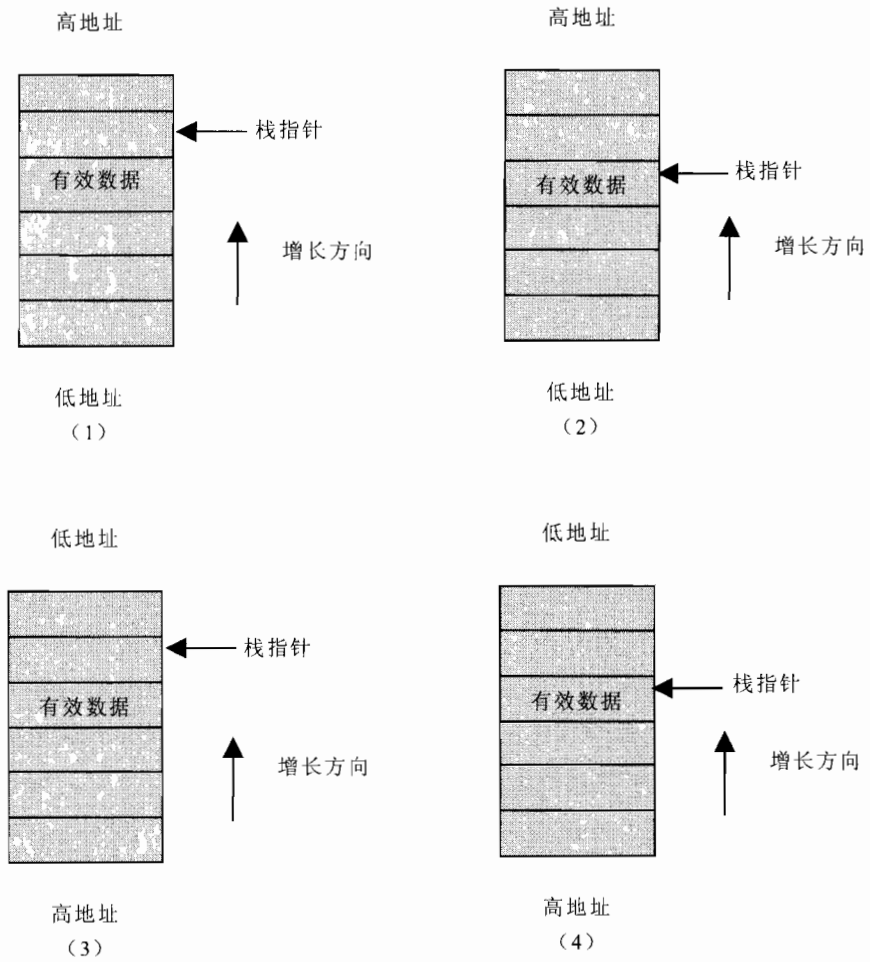


图 3-1　堆栈的四种处理方法

回想一下我们前面的介绍，过程调用标准规定了程序间相互调用时所要遵守的规则，无论程序来源于哪、使用什么语言写成，只要共同遵守某一规则，程序间就可以相安无事、各司其职。也就是说，堆栈工作方式在过程调用标准当中会被明确地规定。AAPCS 中明确指出使用满递减堆栈方式处理堆栈。这表示操作系统在堆栈初始化阶段，必须将堆栈指针寄存器赋予一个高端的内存地址值，在代码 3-2 中，该值是内存的最高地址值减 4。

回到代码 3-2 中，R0 被赋值之后，紧接着程序通过一条 msr 指令更改当前处理器模式为未定义指令模式。在 ARM 的 7 种处理器模式中，除用户模式和系统模式共用一个 R13 寄存器外，其他异常模式都有独立的 R13 寄

存器，这表示各模式的堆栈是彼此独立的。同时，因为不同模式之间的私有寄存器不能够互相访问，因此，想要对不同模式的 R13 寄存器分别进行初始化，必须首先切换到相应模式中，然后再对 R13 寄存器赋值。而 msr 指令便能实现模式切换，其完整格式如下。

```
msr{cond} psr_fields,#immed_8r
```

或

```
msr{cond} psr_fields,Rm
```

显然，代码 3-2 采用的是第一种格式，将一个立即数赋值给 CPSR 寄存器的控制域，其结果是将处理器模式由启动时默认的 SVC 模式切换到未定义模式。在未定义模式中，程序将寄存器 R0 的值赋给堆栈指针寄存器 SP，从而完成了一次异常模式的堆栈初始化工作。接下来，类似的代码被重复执行，最终完成处理器各个模式的堆栈初始化。

在完成堆栈初始化后，代码 3-2 进入下一阶段，清除掉 bss 段的数据。bss 段通常是用来存放程序中未初始化或初始值为零的全局变量或静态变量的一块内存区域。与 data 段不同，bss 段内的所有数据都没有初始值或初始值为零。也就是说，在编译程序的过程中，没有必要将 bss 段的内容生成在可执行程序中，这意味着如果我们将无所谓初始值的变量都空下来不赋值或赋值为零，将会大大减少可执行程序的尺寸，这一点对于对文件大小敏感的程序来说很有意义。当然，有利必有弊，虽然存储空间被节省了，可是这些变量的初始值就不得不在运行时赋值了。换句话说，程序的启动速度会受到一定程度的影响。

有些读者可能会有这样的疑问，对于那些没有初始值的数据，是不是有必要仍然将其初始化成零？其实，这是 C 语言标准所规定的，所有的变量都应该被初始化。具体地说，所有的指针都应该被初始化为空指针，所有的算术类型数据，都应该被初始化成零。正因为如此，即使程序在整个运行期间不会使用零这个值，在操作系统启动的过程中，却仍然需要对 bss 段进行初始化。

于是在代码 3-2 里，程序将 bss 段起始地址和结束地址读出，并算出 bss 段的大小，然后在.clear_loop 循环内，逐个对 bss 段的内容赋值。其中用到的一些汇编指令是前面没有提过的，下面具体介绍一下。

（1）sub 指令是减法指令，几乎所有的体系结构都支持这种指令并使用

同样的指令助记符。我们在这里看到的减法指令稍有变化，是在 sub 后边又多加了一个 s，意思是根据该指令运行的结果来更新 CPSR 寄存器的标志位。减法指令的标准格式是：sub{<cond>}{s} Rd，Rn，N。它可以将寄存器 Rn 减 N 的结果保存在寄存器 Rd 中，并且根据是否带有后缀 s 来选择更改标志位。举例来说，“subs r3,r3,#1”将会使 R3 寄存器的值减一，当减到零时，CPSR 中的零标志位会被置 1。

（2）b 指令是我们早已熟悉的了，但是 b 后边的 gt 后缀我们却是头一次接触。这里的 gt 充当了条件助记符的作用，表示如果“有符号数大于”这一条件发生时便执行跳转。这里所谓的“有符号数大于”针对的是上一条减法指令。无论 Rn 和 N 是正是负，当二者相减的结果大于零时，该条件就会发生。而在代码 3-2 中，这一条件其实是表示 R3 的值并未递减到零，因此需要跳转回.clear_loop 处继续将下一个位置清零，直至 bss 段末尾。

代码 3-2 中有一个细节需要注意，那就是.word 伪指令。这个伪指令可以用来在内存中分配一个 4 字节的空间，并可以同时对其初始化。例如，代码中的“.LC6: .word __stack_end__”，就是在定义空间的同时，又向该空间赋予了__stack_end__这一变量的值，这样就可以直接在程序中引用.LC6 了。

在所有初始化过程完成后，程序跳转到 main 函数处继续接下来的工作。

对于 freeRTOS 来说，程序在进入 main 函数后立即执行 prvSetupHardware 函数，完成特定硬件平台的初始化工作，这些初始化工作因为与平台相关，因此程序本身不具有代表性，但程序执行的基本流程仍然是确定的，总结起来有如下两个方面。

（1）时钟初始化。时钟的频率直接决定了 CPU 的性能，因此如果条件允许，我们都会尽量提高 CPU 的时钟频率，从而提高效率。但是这样，问题也就来了，一般 CPU 内部时钟都是可编程的，时钟的本源往往来源于外部晶振，很多 CPU 允许外接的晶振频率是可选的，而我们必须根据晶振的频率计算出相应的值来产生正确的 CPU 时钟。因此，CPU 通常不是一上电就立刻工作在一个较高的频率下的，而是首先以一个相对较低的频率运行，之后通过编程的方法进行配置，才可以使用高频时钟。除此之外，CPU 内部集成的外围器件或控制器也需要不同频率的时钟，这些时钟频率远远低于 CPU 的工作频率，因此需要通过分频的方法获得。无论怎样，时钟就是机

器的心跳，是不可或缺的。对时钟的初始化几乎是任何一款操作系统初始化阶段的必要步骤。

（2）内存初始化。多数 CPU 与内存的耦合程度并不是特别紧密。通俗地讲，多数 CPU 不需要特别复杂的硬件辅助就可以直接与内存连接。但是，这不能表示已经连接上的内存处于一个可用或者好用的状态。不同的内存芯片、厂商、型号类型在不同的板子上都是不同的，通常我们需要根据实际情况，专门对内存进行配置以适应这种环境。由于内存在没有正确配置之前很可能不能正常使用，或者仅能以很低的效率被使用。因此，初始化内存就成了操作系统初始化时又一必不可少的工作。

除了上述必须完成的工作之外，一款成熟的操作系统还可能包含一些可选的初始化步骤，比如输入输出设备、存储设备的初始化等。只有操作系统赖以生存的环境被正确并且完整的配置后，操作系统才能发挥出它真正的威力。

3.1.3 正式开始写操作系统

前面我们分析了一段基于 ARM 的典型初始化程序并总结了系统初始化的一般步骤，做到了知其然并知其所以然。接下来就利用我们学到的知识和原理，正式开始编写属于我们自己的操作系统。请打开一款代码编辑器，并输入下面这段代码。

代码 3-3

```
.section .startup
.code 32
.align 0
.global _start
.extern __vector_reset
.extern __vector_undefined
.extern __vector_swi
.extern __vector_prefetch_abort
.extern __vector_data_abort
.extern __vector_reserved
.extern __vector_irq
.extern __vector_fiq
```

```
_start:

ldr pc,_vector_reset
ldr pc,_vector_undefined
ldr pc,_vector_swi
ldr pc,_vector_prefetch_abort
ldr pc,_vector_data_abort
ldr pc,_vector_reserved
ldr pc,_vector_irq
ldr pc,_vector_fiq

.align 4

_vector_reset: .word   __vector_reset
_vector_undefined:.word   __vector_undefined
_vector_swi:   .word   __vector_swi
_vector_prefetch_abort:   .word   __vector_prefetch_abort
_vector_data_abort:   .word   __vector_data_abort
_vector_reserved: .word   __vector_reserved
_vector_irq:   .word   __vector_irq
_vector_fiq:   .word   __vector_fiq
```

有些读者可能已经意识到，以前从未接触过或很少接触的汇编程序，现在看起来似乎已非常简单。这说明我们已经在实践中掌握了一定的 ARM 汇编知识。

这段代码我们无须解释，但需要辨析一下不同的跳转指令之间的区别。在 ARM 汇编程序中，要实现程序间跳转有三种方法可供使用：

（1）使用跳转指令 b、bl、bx 等。使用这一系列指令的优点是执行速度快，只需要一个指令周期即可完成跳转。但该系列指令有一个明显的缺点，那就是它们都不能实现对任意地址的跳转。比如说程序在地址 0x0 处运行了跳转指令，但是该指令不能跳转到 0xc0000000 处去运行，这是由 ARM 指令等宽特性决定的。ARM 指令集是 32 位等长的，所以，所有的指令（包括指令的参数在内）都必须在 4 字节的范围内完成。这样，当一条指令需要附带一个立即数或一个地址值作为参数时，该立即数或地址值必然要小于 32 位。事实上，跳转指令 b 所能跳转的最大范围是当前指令地址前后的 32M。

（2）使用内存装载指令，将存储在内存的某一地址装载到程序计数器 PC 中。例如，若我们将跳转的目的地址存储在高于当前地址 24 字节处，就

可以使用“ldr pc，[pc,#24]”这条指令实现跳转。当然，在实际编码中，我们并不一定要亲自做这种偏移量的计算，而可以采用代码 3-3 的方法，用一条 ldr 伪指令来实现。这样，编译器会根据情况将该指令展开成 b 指令或 ldr 指令。ldr 指令能够实现任意地址的跳转，但因为需要读写存储器，所以在执行速度上与 b 指令相比就稍逊一筹了。

（3）第三种跳转方法是通过 mov 指令来实现的。mov 指令是最简单的移动指令，几乎所有的处理器都包含这条指令。通过 mov 指令将已经保存到某一寄存器中的地址值直接赋值给程序计数器 PC，也可以实现程序跳转。使用该方法仍然需要额外的指令或别的手段，将跳转地址预先保存到寄存器中。因此这种方法多用在函数返回时的跳转中。

了解了 ARM 中的跳转方法之后，相信读者已能在编程过程中灵活应用它们了。但是，代码 3-3 是只有当异常模式发生时才会运行的。正常情况下，程序一开始就会马上跳转到__vector_reset 处运行。

下面就让我们跟随程序的执行过程，继续完善我们的操作系统。

代码 3-4

```
.text
.code 32
.global __vector_reset

.extern plat_boot
.extern __bss_start__
.extern __bss_end__

__vector_reset:
    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|SVC_MOD)
    ldr sp,=_SVC_STACK

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|IRQ_MOD)
    ldr sp,=_IRQ_STACK

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|FIQ_MOD)
    ldr sp,=_FIQ_STACK

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|ABT_MOD)
    ldr sp,=_ABT_STACK
```

```
    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|UND_MOD)
    ldr sp,=_UND_STACK

    msr cpsr_c,#(DISABLE_IRQ|DISABLE_FIQ|SYS_MOD)
    ldr sp,=_SYS_STACK

_clear_bss:
    ldr r1,_bss_start_
    ldr r3,_bss_end_
    mov r2,#0x0
1:
    cmp r1,r3
    beq _main
    str r2,[r1],#0x4
    b 1b

_main:
    b plat_boot

_bss_start_:.word   __bss_start__
_bss_end_:.word   __bss_end__

.end
```

在正常模式下，程序会进入__vector_reset 并运行，而后分别进行堆栈寄存器的初始化和 bss 段的清除工作。这里有一个细节需要强调，那就是对__bss_start__和__bss_end__两个变量的定义，在代码 3-4 的开始部分我们用.extern 关键字声明了这两个变量，这表明这些变量来自于别处。但问题是 bss 段的开始和结束的位置是动态的，程序稍有修改就会产生变化，因此我们很难确定一个固定的值。那么这两个值到底是如何确定的呢？

答案是，由编译器确定。回想一下第 2 章关于链接脚本的内容，bss 段、data 段和 text 段的具体位置都是在链接脚本中指定的，也就是说，编译器在编译的时候会计算每段程序的代码、未初始化的全局变量和已经初始化的全局变量，并根据链接脚本安排各段的大小和位置。

读者也许还是会问，就算能够确定 bss 段的起始位置和结束位置，那这两个值又是如何传递的呢？

这个问题的答案还是在于链接脚本，看看下面这段代码，读者朋友们就会明白了。

代码 3-5

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
    .text :
    {
        *(.startup)
        *(.text)
    }
    . = ALIGN(32);

    .data :
    {
        *(.data)
    }

    . = ALIGN(32);
    __bss_start__ = .;
    .bss :
    {
        *(.bss)
    }
    __bss_end__ = .;
}
```

读者可能已经在代码 3-5 中发现了__bss_start__和__bss_end__两个变量。在这段链接脚本中，我们将__bss_start__变量赋予了当前位置值，紧接着的就是 bss 段。因此，__bss_start__变量恰好代表了 bss 段的起始地址，而__bss_end__变量也是通过同样的方式赋值的，最终这两个变量在代码 3-4 中被引用。

回到代码 3-4 中，在对 bss 段的清零工作完成后，代码通过一条跳转指令跳转到函数 plat_boot 处执行，开始外围硬件的初始化工作。

由于操作系统在设计之初就兼顾了自身的可移植性，因此，板载硬件的初始化工作对于不同的平台，虽然毫无相同之处，但却必须要追求一种统一。这种不同平台的统一其实是一种抽象，在程序设计当中，好的抽象能够以最高的效率、通过最少的代码修改实现最大程度的兼容，可以说软件设计能力其实就是一种抽象能力。可能有些读者会觉得我的这段话也很抽象，那就让

我们直接看代码，体会一下抽象软件设计的优势吧。

代码 3-6

```
static init_func init[]={
    arm920t_init_mmu,
    s3c2410_init_clock,
    s3c2410_init_memory,
    s3c2410_init_irq,
    s3c2410_init_io,
    NULL
};

void plat_boot(void){
    load_init_boot(init);
}
```

这段代码其实很简单，首先在程序中定义了一个名为 init、类型为 init_func 的数组。init_func 类型其实是一种函数类型，其定义如下：

代码 3-7

```
typedef void (*init_func)(void);
```

从定义中我们可以看出，init_func 类型的函数既没有参数也没有返回值。初始化函数不需要参数，是因为它们只需要被调用一次，并不需要通过参数的动态改变来执行内容，也不需要返回值，因为这些函数都不允许执行失败，即便能够返回执行失败的结果，在初始化阶段也没有能力补救。但不管怎样，正是 init 数组中的各个成员函数完成了对各种硬件的具体的初始化工作。在 plat_boot 函数中，将 init 数组作为参数传递给了 load_init_boot 函数，而该函数便是这个数组成员函数的实施者，让我们来看一下这个函数的具体实现：

代码 3-8

```
void load_init_boot(init_func *init){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    boot_start();
};
```

函数通过 for 循环依次运行 init 数组中的成员函数，最终调用 boot_start() 函数，至此，操作系统的初始化过程结束。

这段程序之所以这样设计主要是从体系结构相关性这个角度考虑的。代码 3-8 中的函数就是一个与体现结构无关的函数，无论我们的代码运行在什么硬件平台下，load_init_boot 都可以被重复使用。而 init_func 数组是与体系结构相关的，因此当我们需要支持新的体系结构，或者更改原有体系结构的代码时，只需要重写或修改 init_func 数组实现其内部相关函数即可。我们所说的最基本的抽象指的就是这个。将各硬件平台都要实施的部分提炼出来，供所有体系结构复用，并将各平台独有的部分交给平台相关代码随意实现，从而在一定程度上解决了平台间移植的问题。

3.1.4 让启动代码运行起来

目前我们已经完成了操作系统初始化的第一阶段。现在不如让我们暂时停下来，整理并运行一下前边学习到的代码。

首先需要做的是将代码 3-3 的内容保存成文件，命名为 start.s。

为了节省篇幅，在代码 3-4 中我们只列出了核心程序，许多宏定义并没有列出来。为了使这段代码能够运行，我们需要在代码 3-4 最开头的地方添加如下内容：

代码 3-9

```
.equ DISABLE_IRQ,       0x80
.equ DISABLE_FIQ,       0x40
.equ SYS_MOD,           0x1f
.equ IRQ_MOD,           0x12
.equ FIQ_MOD,           0x11
.equ SVC_MOD,           0x13
.equ ABT_MOD,           0x17
.equ UND_MOD,           0x1b

.equ MEM_SIZE,          0x800000
.equ TEXT_BASE,         0x30000000
.equ _SVC_STACK,        (TEXT_BASE+MEM_SIZE-4)
.equ _IRQ_STACK,        (_SVC_STACK-0x400)
.equ _FIQ_STACK,        (_IRQ_STACK-0x400)
```

```
.equ _ABT_STACK,        (_FIQ_STACK-0x400)
.equ _UND_STACK,        (_ABT_STACK-0x400)
.equ _SYS_STACK,        (_UND_STACK-0x400)
```

在代码 3-9 中，程序主要定义了两类宏。

一类是与处理器模式和中断有关的，我们可以使用这些宏来方便地在不同模式间切换，同时使能或禁止中断。这些细节我们会在后续的学习中详细讲解。

另一类宏定义则是与内存有关的。根据 2410 的芯片手册，外部 sdram 可以接在起始地址为 0x30000000 的位置上，我们虚拟的内存便是从这一地址开始的，并假设内存总容量为 8M。这样，我们选取 SVC 模式堆栈的最顶端为内存的最后一个字节（即 0x30000000+0x800000-4），并给其分配 1K 的堆栈空间供该模式使用，其他模式的堆栈空间大小同样为 1K，位置依次排列。我们将代码 3-9 与代码 3-4 的内容保存成文件，命名为“init.s”。

在代码 3-3 中，程序通过.extern 关键字声明了很多外部变量，但这些变量并未定义。于是我们需要新建一个文件并将这些变量的定义写进去，其内容如下：

代码 3-10

```
.global __vector_undefined
.global __vector_swi
.global __vector_prefetch_abort
.global __vector_data_abort
.global __vector_reserved
.global __vector_irq
.global __vector_fiq

.text
.code 32

__vector_undefined:
    nop
__vector_swi:
    nop
__vector_prefetch_abort:
    nop
__vector_data_abort:
    nop
```

```
__vector_reserved:
    nop
__vector_irq:
    nop
__vector_fiq:
    nop
```

这里我们只需要将这段代码复制到文件中，命名为“abnormal.s”。代码3-10还有一条指令，那就是nop指令。其实这条指令在这里根本没有什么作用，只是在执行的时候能够消耗掉一个指令周期。此时我们尚未涉及到异常模式的处理，所以使用nop指令，权当占个位置，在讲到异常模式的时候，我们会进一步改写这些异常处理函数。

为了能够运行这部分程序，我们需要临时将代码3-6、代码3-7和代码3-8改写一下，并使用我们在第2章中编写的helloworld函数来验证操作系统初始化部分的运行进度。

代码3-11

```
typedef void (*init_func)(void);
#define UFCON0  ((volatile unsigned int *)(0x50000020))

void helloworld(void){
    const char *p="helloworld\n";
    while(*p){
        *UFCON0=*p++;
    };
}

static init_func init[]={
    helloworld,
    0,
};

void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
}
while(1);
}
```

在代码 3-11 中，原本是应该填写启动函数的 init 数组，现在只填写了一个 helloworld 函数，目的仅是看到操作系统启动时的运行现象。我们将代码 3-11 保存到文件中，命名为“boot.c”。

接着将代码 3-5 的内容保存到文件中，将其命名为“leeos.lds”。

编译过程非常简单，我们需要写一个 Makefile 文件，将代码 3-12 的内容保存成文件，与源代码放在同一目录下，命名为 Makefile。

代码 3-12

```
CC=arm-elf-gcc
LD=arm-elf-ld
OBJCOPY=arm-elf-objcopy

CFLAGS= -O2 -g
ASFLAGS= -O2 -g
LDFLAGS=-Tleeos.lds -Ttext 30000000

OBJS=init.o start.o boot.o abnormal.o

.c.o:
    $(CC) $(CFLAGS) -c $<
.s.o:
    $(CC) $(ASFLAGS) -c $<

leeos:$(OBJS)
   $(CC) -static -nostartfiles -nostdlib $(LDFLAGS) $? -o $@ -lgcc
    $(OBJCOPY) -O binary $@ leeos.bin

clean:
    rm *.o leeos leeos.bin -f
```

在确认所有文件都处在同一目录下后，只需要在终端上运行 make 这条命令，最终将会生成二进制文件“leeos.bin”。

当然，如果需要在 skyeye 中虚拟运行，还有一个文件是我们需要的，那就是“skyeye.conf”文件，不同于第 2 章的配置文件，这里的 skyeye.conf 的内容稍有变化，如下所示。

```
cpu:  arm920t
mach: s3c2410x

#physical memory
```

```
mem_bank: map=M, type=RW, addr=0x30000000, size=0x00800000,
file=./leeos.bin,boot=yes
mem_bank: map=I, type=RW, addr=0x48000000, size=0x20000000
```

因为在代码中，我们已经假设内存起始于地址 0x30000000 处，大小为 0x00800000。为保证代码能够成功运行，我们必须虚拟出一块内存，起始位置和大小恰与代码一致。

现在万事俱备了，在终端运行 skyeye 命令，将可以看到 helloworld 字符串从 skyeye 中打印出来：

```
……
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
```

看到这个结果，不能不让人兴奋，虽然该结果与第 2 章相比没有什么变化。但从代码实现上看，二者有本质区别，其意义完全不同。我们现在正在一步步地将我们所学的知识转化为实践，逐步打造属于自己的嵌入式操作系统！

当然，现在就沾沾自喜还为时过早。接下来我们就来啃一块硬骨头—— MMU。这一部分内容，相对来说是比较具有挑战性的。如果您的 ARM 基础相对薄弱，或者读过前面的内容之后仍然是一知半解，不妨暂时略过这一节，继续阅读接下来的内容。

3.2 MMU

什么是 MMU 呢？MMU 是 Memory Management Unit 的缩写，它代表集成在 CPU 内部的一个硬件逻辑单元，主要作用是给 CPU 提供从虚拟地址向物理地址转换的功能，从硬件上给软件提供一种内存保护的机制。

举个例子来说，若一个 CPU 没有 MMU，我们想向内存地址 0x0 处写一个整型值 0xffff，就只能使用如下的方法：

```
(int *)0x0=0xffff;
```

但如果一个 CPU 带有 MMU，并且通过软件对 MMU 进行了配置，将地

址 0xc0000000 映射到地址 0x0 处，并激活 MMU。那么当我们想要向内存地址 0x0 处写入一个 0xffff 时，代码应该并且只能是这样的：

```
(int *)0xc0000000=0xffff;
```

这里，我们将映射之前的地址 0x0 称为物理地址，物理地址即硬件自己定义的地址往往是不可更改的，而将映射之后的地址 0xc0000000 称为虚拟地址，这个地址在真实的硬件中并不存在，只不过我们可以通过访问虚拟地址而间接实现对物理地址的访问，并且只能使用这种方法进行访问，如图 3-2 所示。当然，我们也可以实时调整这种映射关系，让别的地址也映射到物理地址 0x0 上，而不一定非得是 0xc0000000。这就是说，这种映射关系允许动态调整，允许多对一的映射。

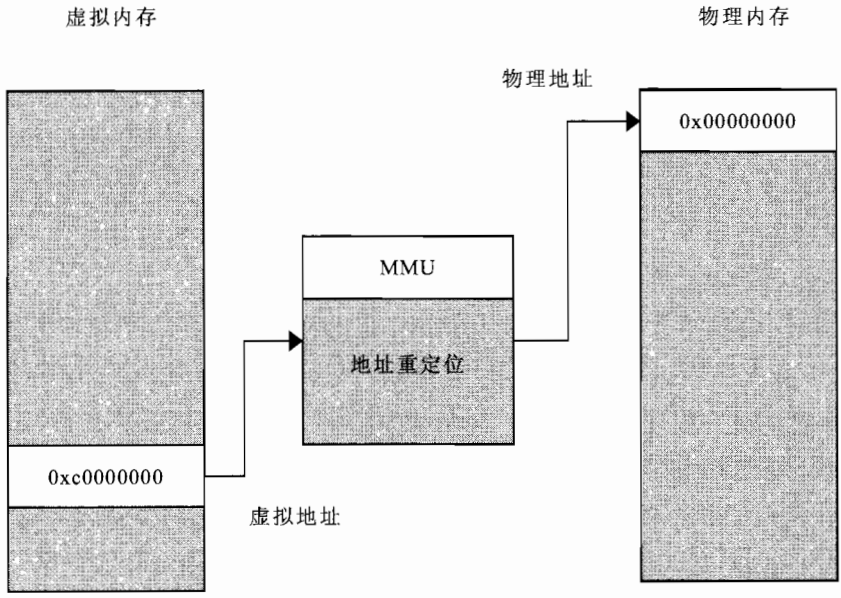


图 3-2　MMU 工作原理

刚刚接触虚拟地址映射的读者朋友们一定都会有这样的疑问，我们为什么要多此一举，非要使用 MMU 进行地址映射呢？

这个问题说来话长，在操作系统还是以单任务的方式运行的年代，MMU 并没有它存在的意义，当操作系统开始支持多任务后，从宏观上看，多个进程并行执行，并且各进程之间的资源都是相互独立的，通过什么手段可以有效地保护各进程的资源不会被其他进程破坏呢？当然，我们可以通过使用软

件划分好各进程的地址空间，但因为各进程仍然有跨界操作的权限，即便在正常执行时进程间彼此不会互相干扰，也无法避免错误执行时的后果。因此就需要从硬件上提供一种机制，彻底限制某个进程对其他进程资源的访问权限，于是内存保护单元应运而生，MMU 能够很好地提供内存保护作用，这就是需要 MMU 的第一个原因。

随着计算机技术的发展，操作系统愈加复杂，应用环境也承现多样化，比如我们经常需要将同一个应用程序同时运行多次，这样，问题就会随之而来。我们知道，一个应用程序在编译完成后，其运行地址也就确定了。假设该程序不可以被重定位，那么编译时所确定的地址就成了程序能够运行的唯一地址，当同一个应用程序被两次运行时，必然造成同一个地址既存储了一个进程的指令或数据又存储了另一个进程的指令或数据，这一矛盾的解决仍然要靠 MMU。例如，一个应用程序在虚拟地址 0x10000000 处存储了一个计数值，当该程序第一次运行时，MMU 将该地址映射到 0x30000000 处，而同一个程序再次被运行时，只需要将 0x10000000 这一地址映射到别的地方，比如 0x31000000，这样，前后两个进程虽然虚拟地址是相同的，但他们却独立运行在不同的物理地址中，因此能够在同一时刻同时运行。给各进程提供独立的地址空间是 MMU 存在的第二个原因。

虽然在高级操作系统中，使用 MMU 优势明显，但同时也存在着很多争议。其中最大的争议是关于地址映射的执行效率问题。尽管整个地址翻译的过程都是由硬件实现的，相比于直接访问地址，时间上的浪费微乎其微，但大量寻址的累积势必会造成系统效率降低。因此很多对实时性具有高要求的操心系统并不倾向于使用 MMU。然而，在一些非实时或对实时性要求并不严格的系统中，MMU 却能够给上层应用程序提供极大的便利。

不管怎样，地址映射这种内存管理方式对于高级操作系统来说仍然是一种普遍采用的方法。所以接下来，就让我们以 ARM 为例，深入研究一下 MMU 的工作原理。

3.2.1 页表

首先让我们来介绍一个概念——页表（page table）。页表就是存储在内存中的一张表，表中记录了将虚拟地址转换成物理地址的关键信息。MMU 正是通过对页表进行查询，实现了地址之间的转换。也就是说，MMU 每次

工作的时候都要去查这张表，从中找出与虚拟地址相对应的物理地址，然后再进行数据存取。页表的作用如图 3-3 所示。

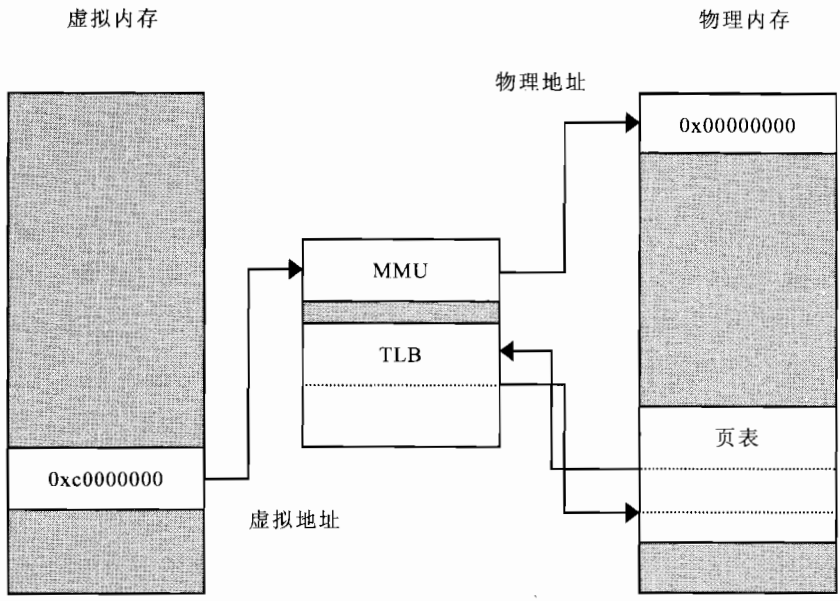


图 3-3　页表的作用

页表中的条目被称为页表项（page table entry），一个页表项负责记录一段虚拟地址到物理地址的映射关系，稍后我们会详细介绍。

既然页表是存储在内存中的，那么程序每次完成一次内存读取时都至少会访问内存两次，相比于不使用 MMU 时的一次内存访问，效率被大大降低了，如果所使用的内存的性能比较差的话，这种效率的降低将会更明显。因此，如何在发挥 MMU 优势的同时使系统消耗尽量减小，就成为了一个亟待解决的问题。

于是，TLB 产生了。TLB 是什么呢？我们叫它转换旁路缓冲器，它实际上是 MMU 中临时存放转换数据的一组重定位寄存器。既然 TLB 本质上是一组寄存器，那么不难理解，相比于访问内存中的页表，访问 TLB 的速度要快很多。因此如果页表的内容全部存放于 TLB 中，就可以解决访问效率的问题了。

然而，由于制造成本等诸多限制，所有页表都存储在 TLB 中几乎是不可能的。这样一来，我们只能通过在有限容量的 TLB 中存储一部分最常用

的页表，从而在一定程度上提高 MMU 的工作效率。

这一方法能够产生效果的理论依据叫做存储器访问的局部性原理。它的意思是说，程序在执行过程中访问与当前位置临近的代码的概率更高一些。因此，从理论上我们可以说，TLB 中存储了当前时间段需要使用的大多数页表项，所以可以在很大程度上提高 MMU 的运行效率。

让我们接着聊页表。页表是由页表项组成的，每一个页表项都能够将一段虚拟地址空间映射到一段物理地址空间中。这里所谓的这段虚拟地址空间，更专业地讲，应该叫页，一个页对应了页表中的一项，页的大小通常是可选的。在 ARM 中，一个页可以被配置成 1K、4K、64K 或 1M 大小（ARM v6 体系以后，不再支持 1K 大小的页），分别叫做微页、小页、大页和段页。页的大小决定了映射的粒度，是根据实际应用有选择地配置的。以 1M 为例，按照我们前面的描述，假设系统中将有 64M 内存需要被映射，那么我们一共需要 64M/1M 个页表项，而每个页表项需要占据 4 个字节，也就是说，有 256 字节的内存要专门负责地址映射，不能用于其他用途。

对于 1K、4K 和 64K 大小的页，MMU 采用二级查表的方法，即首先由虚拟地址索引出第一张表的某一段内容，然后再根据这段内容搜索第二张表，最后才能确定物理地址。这里的第一张表，我们叫它一级页表，第二张表被称为是二级页表。采用二级查表法的主要目的是减小页表自身占据的内存空间，但缺点是进一步降低了内存的寻址效率。不同大小的页对查表方法的支持程度如表 3-1 所示。

表 3-1　不同大小页的查表方法

	页大小			
查表方法	1K	4K	64K	1M
一级查表	不支持	不支持	不支持	支持
二级查表	支持	支持	支持	不支持

下面，首先来研究一下相对简单的一级查表。

3.2.1.1　一级查表

一级查表只支持大小为 1M 的页。准确地讲，这里所谓的 1M 大小的页，

应称为段（section）。此时，一级页表也被称为段页表。图 3-4 描述了段页表的内存分布情况和段页表项的具体格式。

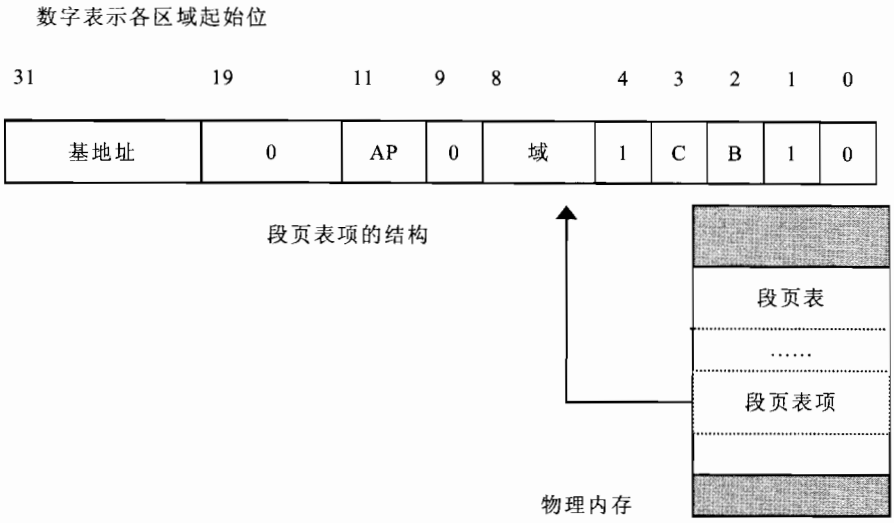


图 3-4　段页表项的结构

段页表中的每一项都类似于图 3-4 中的形式，其中：

（1）31~20 位段表示物理地址的基地址，一共 12 位，也就是说，如果我们通过虚拟地址找到某一个段页表项，那么就可以确定这段虚拟地址所对应的物理地址的高 12 位了。因为段页表项后 20 位正好可以描述 1M 的内存，因此基地址每增加一个单位，物理地址就会增加 1M 空间。所以，我们也可以说该基地址表示了虚拟地址属于哪 1M 范围的物理地址。由此可知，使用段页表进行地址映射时，每一页能够描述 1M 的物理地址空间。进一步讲，段页表最多支持 1024 个页，最多占用系统 4K 字节的内存来存放页表。

（2）11~10 位是 AP 位，区分了用户模式和特权模式对同一个页的不同访问权限。例如，当 AP 位为“11”时，表示任何模式下都可以对该空间进行读写，“10”则表示特权模式可读写该页，而用户模式只能读取该页。对 AP 位详细的描述请参考页权限一节。

（3）8~5 位代表该页所属的域。在 ARM 体系结构中，系统中规定了 16 个域，因此使用 4 个位就能表示该页属于哪个域，而每一个域又有各自独立的访问权限，从而实现了初级的存储器保护。

（4）3 位和 2 位分别代表 cache 和 write buffer。相应的位为 1 则表示被映射的物理地址将使用 cache 或 write buffer。关于 cache 和 write buffer 的有关内容，我们稍后会详述。

（5）1~0 位。这两位用来区分页表类型，对于段页表，这两位的值总是为“10”。

总的来说，段页表项的内容虽然复杂，但归结起来无非就是两个问题，一是，如何解决某一虚拟地址属于哪段物理地址，二是，如何确定这段地址的访问权限。使用其他形式的页表，本质上也是要解决这两个问题。

现在，假设段页表已经被成功地添加到内存之中了，那么接下来的问题是我们应该怎样通过一个虚拟地址找到与之对应的段页表项呢？找到页表项之后，又是怎样找到对应的物理地址的呢？

很显然，虚拟地址本身就可以解决上述问题。

如图 3-5 所示，首先，我们要让 MMU 知道段页表在内存中的首地址，也就是图中所说的页表基地址，因为段页表是我们通过程序确定的，存储在什么位置程序员自然清楚。然后，在 CPU 需要寻址的时候，MMU 就可以自动地利用页表将虚拟地址映射为物理地址并寻址物理地址，其步骤如下：

（1）MMU 取出虚拟地址的前 12 位作为页表项的偏移，结合页表基地址，找到对应的页表项。具体来说，就是将这 12 位数取出，然后左移两位和页表基地址相与，就能得到相应页表项的地址了。例如，页表基地址是 0x10000000，如果虚拟地址是 0x00101000，则前 12 位数是 0x001，那么根据上述步骤将其左移两位，结果为 0x004，那么页表项地址就应该是 0x10000000|0x004=0x10000004，从该地址中读取的 32 位的数据即是该虚拟地址所对应的页表项。

（2）找到与虚拟地址对应的页表项之后，就可以从该页表项中读出一个重要信息，如图 3-4 所示，页表项的前 12 位定位了该虚拟地址所对应的物理地址在哪个范围内。例如，从 0x10000004 中读出的页表项的内容为 0x30000c12，那么我们就已经知道了，虚拟地址 0x00101000 对应的物理地址在 0x30000000 与 0x30100000 之间。既然地址范围已经清楚了，那么虚拟地址又该定位到该范围的哪个位置呢？这就要靠虚拟地址的后 20 位了。

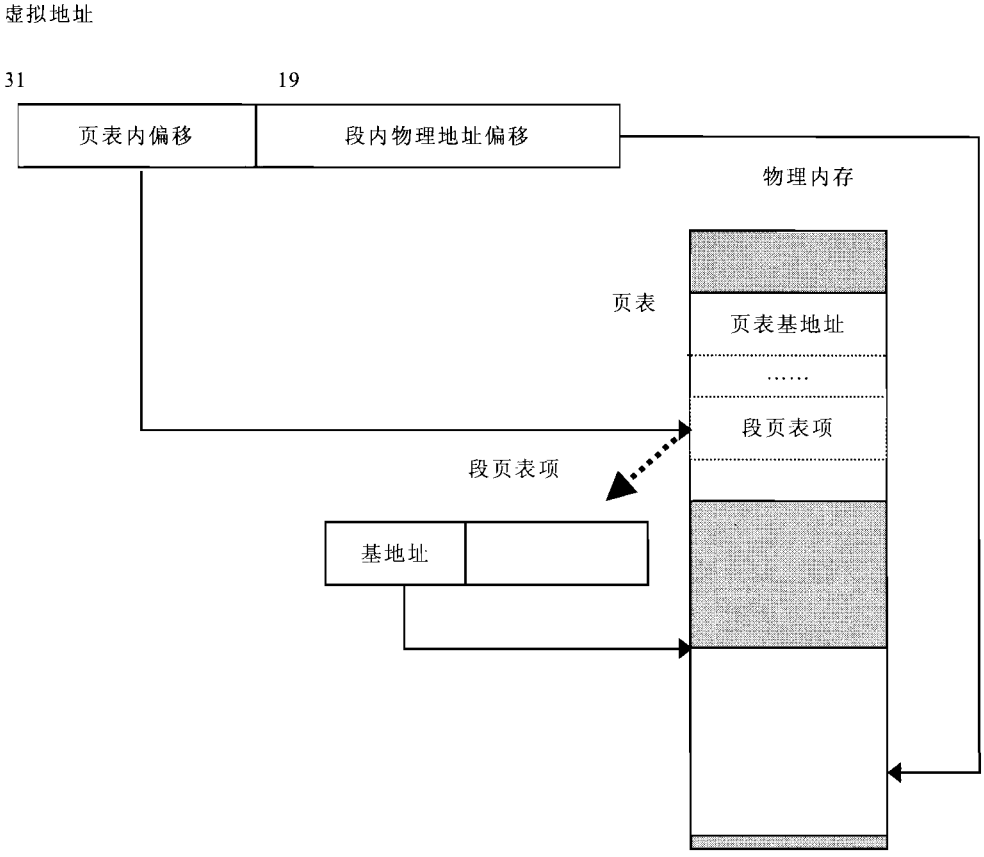


图 3-5　段页表映射过程

（3）MMU 将虚拟地址后 20 位和页表项内容清除掉后 20 位之后的结果做与的操作，就得到了与虚拟地址对应的实际物理地址了。例如，页表项的内容为 0x30000c12，清除掉后 20 位的结果为 0x30000000，虚拟地址的后 20 位为 0x01000，将其和 0x30000000 相与，结果为 0x30001000，这便是最终的物理地址。

当然，MMU 在进行地址映射期间，还要进行访问权限的检查，方法是读出页表项的权限位，按照既定规则去检查，如果允许对该地址进行访问则正常访问，如果不允许访问，则抛出异常，通过程序将其捕获并处理。这便是使用段页表时 MMU 的地址映射过程。

3.2.1.2 二级查表

我们也可以选择使用二级查表的方式去实现地址映射。从原理上讲，一级查表和二级查表其实并没有太大的差别。使用二级查表法，经过一级页表得到的数据不再是记录了物理地址信息的数据了，而是二级页表项的索引信息。而这些信息除了相应位和标志与一级页表项略有不同之外，与一级查表并无不同。

由于在我们的操作系统中没有使用到二级查表法，这部分内容我们就不介绍了，读者如果感兴趣，可以参考相关文档，自己去实现。

3.2.2 页权限

前面我们已经介绍过，内存管理单元不仅能够给进程提供独立的地址空间，而且可以设置进程对某段地址空间的访问权限。总结起来，页的权限主要跟两个方面有关——域（domain）和访问权限（access permision）。

域的权限是以 1M 为单位的，ARM 最多允许定义 16 个不同的域，并允许配置每 1M 的地址空间分属于 16 个域中的任意一个，从而可以通过使用改变域权限的方法来批量控制整块地址空间的访问权限。举个例子来说，假设我们配置系统中的前 4M 内存，使其由第一个域来控制，而将系统中的后 4M 内存进行配置，让第二个域去控制。那么当环境要求让系统中的前 4M 内存可以被自由访问、后 4M 内存只能被特定任务访问时，就可以简单地修改系统中这两个域的权限，迅速地实现要求。在 ARM 920T 体系结构中，这 16 个域的权限是由一个叫做 CP15 的协处理器中的 C3 寄存器来控制的，如图 3-6 所示。

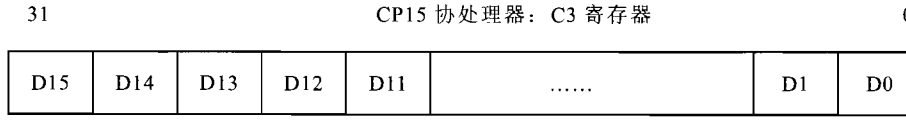


图 3-6 C3 寄存器

在图 3-6 中，在 32 位的寄存器 C3 中，每两位代表一个域权限，一共是 16 个域。而每个域只有 4 种取值，分别是 0b00、0b01、0b10、0b11，这 4

个值分别代表了该域的不同权限类型，如表 3-2 所示。

表 3-2　域权限设置

访问	域权限值	描述
管理者	0b11	该域不受来自页表的权限限制，总可以被访问
保留	0b10	结果不可预料
用户	0b01	访问受页表中的权限限制
禁止访问	0b00	将产生域错误

因此，只要配置好某块内存对应域的权限，就可以实时地改变这块内存区的访问类型。接下来的问题是如何知道哪块内存属于哪个域呢？读者只要回头看一下我们前面讲的段页表项的内容就会明白了。在段页表项中，第 8 位~第 5 位的值用来表示该段页表项属于 16 个域中的哪一个。MMU 在进行地址映射时，需要读出这些位，在 CP15：C3 中找到对应域的权限，根据表 3-2 中的表述确定该段内存的访问权限，然后进行访问，域就是通过这种方式实现了一种粗粒度的页权限。

另一种更有效的权限控制方法是访问权限（access permision），段页表项中 11 位和 10 位代表该段页表项的 AP 位，其可能的取值也只有 4 种，与之对应的读写权限如表 3-3 所示。

表 3-3　AP 权限设置

AP 位	S 位	R 位	特权模式	用户模式
0b11	忽略	忽略	可读写	可读写
0b10	忽略	忽略	可读写	只读
0b01	忽略	忽略	可读写	不可访问
0b00	0	0	不可访问	不可访问
0b00	0	1	只读	只读
0b00	1	0	只读	不可访问
0b00	1	1	结果不可预料	结果不可预料

表 3-3 中的 S 位和 R 位指的是协处理器 CP15：C1 寄存器中的两个位，专门负责全局修改存储器访问权限，正确地配置好这两个位能够提高系统访问大块内存时的速度。设定 S 位能够使用户任务无法对页进行访问，同时又允许了特权模式的读权限，而改变 R 位则可以允许特权模式和用户模式都对页有只读的权限。当然，R 位和 S 位能够发挥作用的前提是页表项中的 AP 位必须为零值。当页表项中的 AP 位为非零值时，表示该页表项所对应的内存权限由 AP 位的值来决定，当 AP 为零，则该页表项所代表的内存的

读写权限由 R 位和 S 位决定。修改 R 位和 S 位要比一项一项地去修改页表项更迅速，因此适用于对大块内存进行权限控制。

3.2.3 cache 和 write buffer

如果有人问，从 CPU 的角度来看哪种存储器件的访问速度最快？答案应该是寄存器，因为寄存器是寄生在 CPU 芯片内部的，与 CPU 处理单元距离最近，速度最快。比寄存器稍慢一些的呢？那就应该是各种形式的内存了。有些 CPU 内部会内置小容量的静态内存，但多数内存是工作在板卡的层次上的，也就是说，多数情况下，内存与 CPU 需要通过主板进行连接，再加上廉价的内存本身就有读写速度的瓶颈，这样一来，站在 CPU 的角度上看，内存也就变成一个慢速设备了。因此，我们需要一种有效手段来提高内存的访问效率，解决的方法就是 cache 和 write buffer。

简单地说，cache 和 write buffer 都是内置于 CPU 内部的一小段高速存储器，cache 中保存着最近一段时间被 CPU 使用过的内存数据，而 write buffer 则是用来应对内存的写操作的，将原本要写向内存的数据暂写到 write buffer 中，等到 CPU 空闲的时候，数据才会慢慢地被搬移到内存里。

由于成本的问题，cache 和 write fuffer 的容量都不会很大，因此，里边只能保存局部的数据信息。但这也不会影响 cache 和 write buffer 发挥其提高系统性能的作用。还记得前面提到的存储器访问的局部性原理吗？程序在一段时间内对数据的访问总是局部的。所以大多数情况下，cache 和 write buffer 会对程序的运行速度有很大的帮助。在这里我们只需要介绍一下 cache 和 write buffer 在页表项中的配置方法，而这足以使我们建立一个段页表了。

在段页表项中，第 3 位和第 2 位分别用来设置 cache 和 write buffer。简单地说，当这两个位为 1 时，表示该段页表项所描述的内存将使用 cache 和 write buffer，如果将这两个位设置为 0，则表示禁用 cache 和 write buffer。

当我们按照上述方法完成了段页表的初始化工作后，就可以使能 MMU 了，这主要是通过配置 ARM 的 CP15 协处理器实现的。

可能有些读者看着前面冗长的描述已经厌倦了，此时迫不及待地想实践一下。那我们就先看一段代码，体会一下从程序上如何初始化段页表。

代码 3-13

```
#define PAGE_TABLE_L1_BASE_ADDR_MASK        (0xffffc000)
#define VIRT_TO_PTE_L1_INDEX(addr)     ((addr)&\
                                        0xfff00000)>>18)
#define PTE_L1_SECTION_NO_CACHE_AND_WB  (0x0<<2)
#define PTE_L1_SECTION_DOMAIN_DEFAULT       (0x0<<5)
#define PTE_ALL_AP_L1_SECTION_DEFAULT       (0x1<<10)
#define PTE_L1_SECTION_PADDR_BASE_MASK      (0xfff00000)
#define PTE_BITS_L1_SECTION                 (0x2)
#define L1_PTR_BASE_ADDR                   0x30700000
#define PHYSICAL_MEM_ADDR            0x30000000
#define VIRTUAL_MEM_ADDR                    0x30000000
#define MEM_MAP_SIZE                        0x800000
#define PHYSICAL_IO_ADDR                   0x48000000
#define VIRTUAL_IO_ADDR                    0xc8000000
#define IO_MAP_SIZE                            0x18000000

unsigned int gen_l1_pte(unsigned int paddr){
    return (paddr&PTE_L1_SECTION_PADDR_BASE_MASK)|\
                        PTE_BITS_L1_SECTION;
}

unsigned int gen_l1_pte_addr(unsigned int baddr,\
                                    unsigned int vaddr){
    return (baddr&PAGE_TABLE_L1_BASE_ADDR_MASK)|\
                                VIRT_TO_PTE_L1_INDEX(vaddr);
}

void init_sys_mmu(void){
    unsigned int pte;
    unsigned int pte_addr;
    int j;

    for(j=0;j<MEM_MAP_SIZE>>20;j++){
        pte=gen_l1_pte(PHYSICAL_MEM_ADDR+(j<<20));
        pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;
        pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;
        pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;
        pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\
                                VIRTUAL_MEM_ADDR+(j<<20));
        *(volatile unsigned int *)pte_addr=pte;
    }
```

```
    for(j=0;j<IO_MAP_SIZE>>20;j++){
        pte=gen_l1_pte(PHYSICAL_IO_ADDR+(j<<20));
        pte|=PTE_ALL_AP_L1_SECTION_DEFAULT;
        pte|=PTE_L1_SECTION_NO_CACHE_AND_WB;
        pte|=PTE_L1_SECTION_DOMAIN_DEFAULT;
        pte_addr=gen_l1_pte_addr(L1_PTR_BASE_ADDR,\
                                 VIRTUAL_IO_ADDR+(j<<20));
        *(volatile unsigned int *)pte_addr=pte;
    }
}
```

代码 3-13 中使用的两个核心函数分别是 gen_l1_pte 和 gen_l1_pte_addr，gen_l1_pte 函数以物理地址为参数，能够返回与该物理地址相对应的段页表项的内容，函数直接返回了下面这个表达式的值：

```
(paddr&PTE_L1_SECTION_PADDR_BASE_MASK)|\
PTE_BITS_L1_SECTION
```

PTE_L1_SECTION_PADDR_BASE_MASK 的值是 0xfff00000，和任意一个物理地址相与，所得到的即是该物理地址的段地址。换句话说，这个结果恰好代表了该物理地址属于哪一段内存范围，而这正是段页表项的内容之一。PTE_BITS_L1_SECTION 被定义为 0x2，根据我们前面的描述，页表项的后两位如果是 0b10 则表示该页表项为段页表项。因此，该函数返回的正是一个基本的段页表项内容。而函数 gen_l1_pte_addr 则是通过段页表的基地址与虚拟地址产生该段页表项的地址，其表达式如下：

```
(baddr&PAGE_TABLE_L1_BASE_ADDR_MASK)|\
VIRT_TO_PTE_L1_INDEX(vaddr);
```

PAGE_TABLE_L1_BASE_ADDR_MASK 被定义为 0xffffc000，与页表基地址相与，确保了当我们传递不正确的页表基地址时，迫使该页表基地址的后 14 位是零。这样做的原因何在？回顾一下虚拟地址到物理地址的映射过程，MMU 需要将虚拟地址的前 12 位作为段页表项地址的偏移量，结合段页表基地址得到真正的段页表项的地址，而又因为段页表项为 4 个字节，需要两位表示，因此一共需要 14 位来表示段页表项的偏移，那么段页表基地址的后 14 位就应该是零了。当计算某一页表项地址时，该地址的前 18 位即为页表基地址，14 位~2 位应该是虚拟地址的前 12 位，而最后两位总是零。宏 VIRT_TO_PTE_L1_INDEX 的作用正是计算偏移量，再和段页表基地址做

与操作，就能得到该段页表项的物理地址了。

有了这两个函数，再加上设置域、AP 以及 cache 和 write buffer 的宏，即可实现段页表项的初始化。函数 init_sys_mmu 通过一个循环操作将物理地址 0x30000000~0x30800000 映射到虚拟地址的 0x30000000~0x30800000 处，这样一来，就正好覆盖了我们虚拟出来的 8M 内存。采用这种映射方法，很大程度上简化了代码的复杂度。

既然操作系统已经使用了 MMU，那么在编译系统源代码时就必须使用虚拟地址链接。倘若虚拟地址和物理地址不一致，那么操作系统开始运行的第一个工作就是通过虚拟地址计算物理地址，从中取出页表项并迅速激活 MMU，其复杂程度可想而知。

某些操作系统，比如 Linux，因为有跨平台支持的需求，很多概念都被抽象化了，Linux 系统就规定内核默认使用虚拟地址空间 3~4G 的范围，而很少会有某种 CPU 会将物理内存映射到 3G 之后的地址。因此，对于 Linux 来说，物理内存地址与虚拟内存地址的不一致就在所难免了。

我们的操作系统在设计时，也在一定程度上考虑到了跨平台兼容的问题。当然，这并不意味着我们必须要把物理内存地址和虚拟内存地址分开。关于操作系统的设计问题，我们不急于在这里探讨。读者只需要认识到，将内存所在的物理地址对等地映射给虚拟地址是比较好的选择。当然，除了内存所在地址之外的其他地址应该如何被映射就无所谓了。

为了说明并验证 MMU 的工作原理，在代码 3-13 中，我们将 0x48000000~0x60000000 的物理地址范围映射到虚拟地址 0xc8000000~0xe0000000 处，熟悉 s3c2410 的读者朋友应该知道，s3c2410 的外设地址空间就处在这一范围内。因此，当 MMU 被激活以后，我们就不能通过将数据写入到 0x50000020 来显示数据了，而必须要将显示的数据写入到 0xd0000020 才可以。

现在，只要激活 MMU，我们的系统就可以在虚拟地址中工作了。

3.2.4　激活 MMU

MMU 的配置和控制都是通过操作 CP15 协处理器实现的。ARM 支持 16 个协处理器，当然，并不是所有的 ARM 系统结构都全部包含这些协处理器。在 ARM920T 系列处理器中，协处理器只有两个，一个是负责系统调试

的 CP14，另一个就是负责系统控制的 CP15。系统中的 cache、write buffer、MMU、时钟模式等都可以通过操作 CP15 来完成。

访问 CP15 协处理器不能使用常规指令，ARM 提供了一组专门操作协处理器的指令，这些指令与 ARM 自身的指令格式有很大的不同。

协处理器也有属于它自己的寄存器，比如 CP15 协处理器内部就有 16 个寄存器，通常使用 C*n* 来表示，这里的 *n* 代表协处理器寄存器的序号。想要使能 MMU，我们所要做的便是正确地配置 CP15 的相关寄存器。

让我们来看一下这段代码：

代码 3-14

```
void start_mmu(void){
    unsigned int ttb=L1_PTR_BASE_ADDR;
    asm(
        "mcr p15,0,%0,c2,c0,0\n"
        "mvn r0,#0\n"
        "mcr p15,0,r0,c3,c0,0\n"
        "mov r0,#0x1\n"
        "mcr p15,0,r0,c1,c0,0\n"
        "mov r0,r0\n"
        "mov r0,r0\n"
        "mov r0,r0\n"
        :
        : "r" (ttb)
        : "r0"
    );
}
```

有些读者读了上述代码，可能会觉得很奇怪，这明明是一个 C 函数，为什么其中又包含了汇编代码呢？其实，这里我们使用的是 GCC 内联汇编的技术。使用 C 代码内联汇编的编程方式在某些情况下可以发挥出强大的作用，如手动优化软件关键部分的代码、直接调用特殊的处理器指令实现特定功能等。

在针对 ARM 体系结构的编程中，我们很难直接使用 C 语言产生操作协处理器的相关代码，因此使用汇编语言来实现就成为了唯一的选择。但由于对 MMU 的操作也涉及相对复杂的逻辑，如果完全通过汇编代码实现，又会过于复杂、难以调试。这样看来，C 语言内嵌汇编的方式倒是一个不错的选择。

然而，使用内联汇编的一个主要问题是，内联汇编的语法格式与使用的编译器直接相关，也就是说，使用不同的 C 编译器内联汇编代码时，它们的写法是各不相同的。为了让读者读懂代码 3-14 的内容，我们首先需要介绍一下在 ARM 体系结构下 GCC 的内联汇编方法。

3.3　GCC 内联汇编

首先，让我们来共同了解一下 GCC 内联汇编的一般格式：

```
asm(
    代码列表
    : 输出运算符列表
    : 输入运算符列表
    : 被更改资源列表
);
```

在 C 代码中嵌入汇编需要使用 asm 关键字，在 asm 的修饰下，代码列表、输出运算符列表、输入运算符列表和被更改的资源列表这 4 个部分被 3 个 “:” 分隔。这种格式虽然简单，但并不足以说明问题，我们还是通过例子来理解它吧。

代码 3-15

```
void test(void){
    ……
    asm(
        "mov r1,#1\n"
        :
        :
        :"r1"
    );
    ……
}
```

在代码 3-15 中，函数 test 中内嵌了一条汇编指令实现将立即数 1 赋值给寄存器 R1 的操作。由于没有任何形式的输出和输入，因此输出和输入列表的位置上什么都没有填写。但是，我们发现在被更改资源列表中，出现了

寄存器 R1 的身影，这表示我们通知了编译器，在汇编代码执行过程中 R1 寄存器会被修改。

寄存器被修改这种现象发生的频率还是比较高的。例如，在调用某段汇编程序之前，寄存器 R1 可能已经保存了某个重要数据，当汇编指令被调用之后，R1 寄存器被赋予了新的值，原来的值就会被修改，所以，需要将会被修改的寄存器放入到被更改资源列表中，这样编译器会自动帮助我们解决这个问题。也可以说，出现在被更改资源列表中的资源会在调用汇编代码一开始就首先保存起来，然后在汇编代码结束时释放出去。所以，代码 3-15 与如下代码从语义上来说是等价的。

代码 3-16

```
void test(void){
    ……
    asm(
        "stmfd sp!,{r1}\n"
        "mov r1,#1\n"
        "ldmfd sp!,{r1}\n"
    );
    ……
}
```

代码 3-16 中的内联汇编既无输出又无输入，也没有资源被更改，只留下了汇编代码的部分。由于程序在修改 R1 之前已经将寄存器 R1 的值压入了堆栈，在使用完之后，又将 R1 的值从堆栈中弹出，所以，通过被更改资源列表来临时保存 R1 的值就没什么必要了。

在以上两段代码中，汇编指令都是独立运行的。但更多的时候，C 和内联汇编之间会存在一种交互。C 程序需要把某些值传递给内联汇编运算，内联汇编也会把运算结果输出给 C 代码。此时就可以通过将适当的值列在输入运算符列表和输出运算符列表中来实现这一要求。请看下面的代码：

代码 3-17

```
void test(void){
    int tmp=5;
    asm(
        "mov r4,%0\n"
```

```
        :
        : " r " (tmp)
        : " r4 "
    );
}
```

代码 3-17 中有一条 mov 指令，该指令将%0 赋值给 R4。这里，符号%0 代表出现在输入运算符列表和输出运算符列表中的第一个值。如果%1 存在的话，那么它就代表出现在列表中的第二个值，依此类推。所以，在该段代码中，%0 代表的就是 " r " (tmp)这个表达式的值了。

那么这个新的表达式又该怎样解释呢？原来，在 " r " (tmp)这个表达式中，tmp 代表的正是 C 语言向内联汇编输入的变量，操作符 " r " 则代表 tmp 的值会通过某一个寄存器来传递。在 GCC4 中与之相类似的操作符还包括 " m " 、 " I " ，等等，其含义见表 3-4。

表 3-4　嵌入汇编操作符节选

操作符	含义
r	通用寄存器 R0~R15
m	一个有效内存地址
I	数据处理指令中的立即数
X	被修饰的操作符只能作为输出

从结构上讲，代码 3-17 与代码 3-14 已经很类似了。与输入运算符列表的应用方法一致，当 C 语言需要利用内联汇编输出结果时，可以使用输出运算符列表来实现，其格式应该是下面这样的。

代码 3-18

```
void test(void){
    int tmp;
    asm(
        " mov %0,#1\n "
        : " =r " (tmp)
        :
    );
}
```

在代码 3-18 中，原本应出现在输入运算符列表中的运算符，现在出现在了输出运算符列表中，同时变量 tmp 将会存储内联汇编的输出结果。这

里有一点可能已经引起大家的注意了，代码 3-18 中操作符 r 的前面多了一个“=”。这个等号被称为约束修饰符，其作用是对内联汇编的操作符进行修饰。几种修饰符的含义如表 3-5 所示。

表 3-5　GCC4 中嵌入汇编修饰符

修饰符	说明
无	被修饰的操作符是只读的
=	被修饰的操作符只写
+	被修饰的操作符具有可读写的属性
&	被修饰的操作符只能作为输出

当一个操作符没有修饰符对其进行修饰时，代表这个操作符是只读的，这正好符合代码 3-17 的要求。但是在代码 3-18 中，我们需要将内联汇编的结果输出出来，那么至少要保证该操作符是可写的。因此，“=”或者“+”也就必不可少了。

至此，本书对 ARM 体系结构下 GCC 内联汇编方法的讲解就算完成了。当然，出于篇幅的限制，GCC 内联汇编还有很多细节我们没有介绍。这些细节或者不常用，或者不重要，总之，在掌握了前面介绍的基本知识的前提下，我们可以在需要这些细节的时候查阅相关参考手册，以便进一步解决更复杂的问题。

现在让我们再次回到代码 3-14，你会发现，代码 3-14 中原本晦涩难懂的内联汇编代码，现在看起来就非常简单了。接下来我们就来分析一下这段汇编代码，看看 ARM 的协处理器是如何操作的。

首先，我们要学习两条指令，分别是 mcr 和 mrc，这两条指令都是跟协处理器的操作有关的，它们能够实现协处理器寄存器与通用寄存器之间的数据传递，其格式如下：

```
<mrc|mcr>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
```

其中，mrc 指令能够将协处理器寄存器中的内容传输到 ARM 通用寄存器中，而 mcr 则实现了相反的操作，将通用寄存器中的值传递给协处理器中的寄存器，“{}”内的表达式是可选的。cp 代表协处理器，不同的协处理器的作用各不相同，如 CP15 恰与系统控制有关；opcode1 被称为第一操作数；Rd 是 ARM 通用寄存器；Cn、Cm 代表的是协处理器的寄存器；opcode2 是第二操作数。整个指令看起来有些复杂，这么多的操作数和寄存器，在实

际使用中应该如何选择呢？这一点读者不需要担心，在实际操作中应该选择什么样的操作数，使用哪些协处理器寄存器，这些都是由协处理器所规定的，我们只需要依照其格式使用即可。

好了，结合这两条协处理器指令以及 GCC 下内联汇编的技术，读懂代码 3-14 也就轻而易举了。下面让我们具体来分析一下这段代码。

```
mcr p15,0,%0,c2,c0,0
```

这是在代码 3-14 中出现的第一条汇编指令，其中%0 的值为 L1_PTR_BASE_ADDR。当然这个值最终会通过寄存器传递，所以上面的指令等效于如下两条指令：

```
mov r0,#L1_PTR_BASE_ADDR
mcr p15,0,r0,c2,c0,0
```

很显然，这将会把 L1_PTR_BASE_ADDR 这个值传递给 CP15 协处理器中的 C2 寄存器，而 C2 寄存器正是负责保存页表基地址的。回想一下，前面提到的地址映射过程，MMU 在进行地址映射时，首先需要拿到页表的基地址，然后通过虚拟地址的部分位找到以页表基地址为基础的页偏移地址，读出页表项，然后定位到物理地址的某个区段，最后再结合虚拟地址剩下的位找到对应的物理地址。这个页表基地址就保存在 CP15 协处理器的 C2 寄存器中，也就是说激活 MMU 的第一步工作首先就应该是初始化页表基地址。

在完成对页表基地址的初始化工作之后，第二步的工作就是处理好页权限的问题。

```
mvn r0,#0
mcr p15,0,r0,c3,c0,0
```

回忆一下页权限一节，页表的正确设置只是成功使用 MMU 的一个方面，另一方面，我们必须保证该段内存具有足够的读写权限，而读写权限首先是由域来控制的，如果域权限允许页表项的权限发挥作用，则由页表项中的权限位来决定页的权限。

在这里为了简化步骤，我们把所有 16 个域权限都设置成 0b11，允许所属的页自由访问。而这 16 个域权限恰是由 CP15 协处理器中的 C3 寄存器负责的。细心的读者可能注意到了，代码中我们使用了一条新的指令 mvn，

光论长相，这条指令与 mov 类似，其实，二者的功能也是近似的。mvn 指令首先将目的操作数取反，然后再赋值给原操作数，所以，mvn r0,#0 的结果是将 0xffffffff 赋值给 R0。最后，mcr p15,0,r0,c3,c0,0 使得 C3 寄存器的值也为 0xffffffff。这样，系统中的 16 个域权限就全部设置成了 0b11。

到这里，我们已经做好了激活 MMU 前的所有准备工作，已经可以正式激活 MMU 了。

在代码 3-14 中，MMU 的激活是通过下面这两条指令来实现的。

```
mov r0,#0x1
mcr p15,0,r0,c1,c0,0
```

这两条指令的目的是将 CP15 协处理器的 C1 寄存器中的第一位置为 1，而这一位，掌管了 MMU 激活的大权。一旦 C1 寄存器的第一位被设置成 1，MMU 便被激活了，从这一时刻开始，物理地址便与世隔绝。

最后我们还需要几条空操作，目的是将激活 MMU 之前流水线上的指令清空。关于 ARM 流水线的故事，我们会在后续章节中相继介绍。

读到这里，有的读者可能会产生这样一种感受，我们似乎在对 ARM 还是一知半解的情况下就进行了相当多的程序开发。程序员在涉足一个全新的领域时，其开发过程大多也是这样的，我们几乎没有足够的时间成本，在完全掌握一个新领域之后再去进行开发。从另一个角度来讲，如果我们将有关 ARM 的所有内容都先介绍完之后再介绍操作系统的实现，那这本书可能就会变成 ARM 体系结构与应用之类的书了。这样一来，枯燥不说，也会大大削减大家的学习热情。而本书这种精心设计的内容布局会让读者在不知不觉中就彻底地掌握了原本枯燥无味的知识。不管怎样，有效的学习方法是可以让学习事半功倍的。

我们回来再看一下代码，你会发现针对 MMU 的操作已经彻底完成了。

现在，让我们以之前的启动代码为基础来运行一下这段代码。

首先将代码 3-13 与代码 3-14 的内容保存成文件，命名为“mmu.c”。然后修改原有的 Makefile 文件，将下面的这条代码：

```
OBJS=init.o start.o boot.o abnormal.o
```

更改为：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o
```

修改原有的“boot.c”文件，修改其中的 plat_boot 函数如下：

代码 3-19

```
void plat_boot(void){
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    while(1);
}
```

代码 3-19 中新添加了三个函数，其中，init_sys_mmu 和 start_mmu 两个函数的具体实现我们已经介绍过了。在 start_mmu 函数运行之后，MMU 被激活，然后我们可以利用函数 test_mmu 来验证一下关于 MMU 这部分代码的正确性，具体实现如下：

代码 3-20

```
void test_mmu(void){
    const char *p="test_mmu\n";
    while(*p){
        *(volatile unsigned int *)0xd0000020=*p++;
    };
}
```

该函数的实现很简单，只是将字符串 " test_mmu\n " 依次写入地址 0xd0000020 处。请回头看一下代码 3-13，我们将物理地址从 0x48000000 开始的一段线性区域映射到虚拟地址 0xc8000000 处。也就是说，代码 3-20 中的虚拟地址 0xd0000020 实际上对应着物理地址 0x50000020，而这个地址正是 2410 的串口 FIFO 寄存器。

从前面的例子中我们已经知道，在 skyeye 虚拟环境下串口是作为默认终端存在的，写向串口中的数据都将输出到标准输出中去，也就是说，字符串 " test_mmu\n " 最终会打印到终端当中去。而与原来的方法不同的是，我们这里是在 MMU 被激活的前提下使用虚拟地址进行操作的，从而验证了前面关于 MMU 这部分代码的正确性。

好了，现在将代码 3-20 中的内容添加到文件“boot.c”中。在终端运行

make 命令，如果一切正常的话，新的“leeos.bin”文件就会生成。在终端运行 skyeye 命令，你将看到如下输出结果：

```
……
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
```

字符串 test_mmu 被打印到终端中，MMU 被成功激活了！

3.4 总结

本章我们主要讨论了操作系统启动时的初始化问题。从理论上探讨了操作系统启动的一般步骤。不过理论终归是理论，一切理论只有应用到实际当中才能真正发挥作用，理论联系实际也是本书秉承的原则，因此，本章中包含有大量的练习代码，这些代码密切结合理论，同时突出重点、通俗易懂，并配有详细的解释。另外，本章也展开了对堆栈理论、系统设计模式等内容的描述。

本章还包含了一个操作系统的难点问题，即 MMU 的问题。在这部分内容当中，我们使用了大量的篇幅描述了 MMU 的原理和作用，然后讲解了在 ARM 中虚拟地址到物理地址映射的过程，并就 MMU 与系统控制的重点问题进行了介绍。期间还讲解了 ARM 协处理器的操作方法、GCC 内联汇编技术等重要内容。这些内容五花八门，涉及到了计算机原理的方方面面，讲述的内容虽然重要却很容易被忽视。本章结合大量实例对上述内容做了系统的阐释，使原本晦涩难懂的知识点变得易于理解，当然，对于庞大的操作系统理论来说，这些知识也仅仅是冰山一角，接下来还有更多的内容等待我们去学习。